Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

Xiaoke Li
June-03-2023

Code Generation

Project Supervisor: Shoaib Jameel

Second Examiner: Leonardo Aniello

A project progress report submitted for the award of
Bsc Computer Science

Abstract

In this study, we introduce a method for fine-tuning pre-trained language models using reinforcement learning and critic models to generate syntactically and semantically correct code. Our approach leverages unit test signals and structural alignment feedback to provide granular guidance during code generation. By modeling program synthesis as a reinforcement learning problem, we employ a policy network for token generation and a critic model to predict unit test outcomes, offering intermediate rewards. This method improves traditional policy gradient updates and demonstrates significant improvements in code quality. Future work will explore multi-task learning, more complex rewards, online learning, and real-world applications.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

I have not used any resources produced by anyone else.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

ECS Statement of Originality Template, updated August 2018, Alex Weddell

Contents

1 Introduction **5**

2 Background Literature **8**

2.1 Pre-AI and AI Winter Period 9

2.2 The Rise of AI Period 11

3 Lyrics Generation - Methodology **14**

3.1 Lyrics generation task based on LSTM 14

4 Code Generation - Methodology **22**

4.1 Program Synthesis Task 23

4.2 Reinforcement Learning Framework 24

4.3 Model 26

4.4 Data 27

4.4.1 Data collection 28

4.4.2 Data Cleansing and Pre-processing 29

4.4.3 Data Analysis 29

4.5 Enhanced Reward Function 30

4.5.1 Compiler Signal 30

4.5.2 Syntactic Matching Score 31

4.5.3 Semantic Matching Score 32

4.6 Critic Model Enhancements 32

5 Code generation - Evaluation 36

6 Conclusion and Future Work 38

7 Project Planning 40

Chapter 1

Introduction

Programming has undergone a transformative evolution since its inception. One of the earliest instances of a machine employing a form of programming was the Jacquard loom, operational in the early 19th century. This machine utilized punched cards to control its operations, a rudimentary yet groundbreaking approach to programming (26). The concept of programming has significantly evolved from these primitive origins to what is now recognized as computer programming, a complex domain focusing on the development of computer software (11). This evolution has not only revolutionized the tools and techniques employed in programming but has also greatly expanded its accessibility and application. Currently, programming is increasingly considered a fundamental interdisciplinary skill that transcends the traditional boundaries of computer science and engineering. This shift is evidenced by the integration of programming education across all levels of educational curricula, a significant departure from its previous confinement to higher education sectors (14). As the scope of computer programming continues to expand into new areas, there is a corresponding need to develop tools that cater not only to the needs of experienced developers but also to those new to this essential skill. This ongoing development is critical to ensuring that programming remains a dynamic and accessible discipline in the face of rapidly advancing technological landscapes.

As programming has transitioned from a specialized skill confined to computer experts to a ubiquitous language spanning multiple fields and academic levels (6), its widespread adoption has not only broadened the scope of technology applications but also intensified the demand for innovative tools (5), even foster a deeper interdisciplinary understanding and application (12). The democratization of programming means that the requirement for programming skills now extends beyond traditional computer science domains to various non-traditional fields, for instance, in the realm of healthcare, automated clinical coding systems have transformed medical documentation (30), underscoring the importance of developing tools that support this diverse applicability. With the increasing complexity of programming tasks, modern developers, including those utilizing large language models (LLMs) (24), require robust and versatile tools to navigate the evolving technological challenges. Consequently, the development of efficient tools that cater to these needs is not merely necessary but critical for driving technological innovation and applications.

Despite the enduring challenges inherent in the field of code synthesis, the convergence of software engineering and machine learning has ushered in an era where automated code generation is no longer a futuristic concept but a tangible reality. Recent years have witnessed a significant surge in the use of deep learning and neural language models (LMs) to automate complex software engineering tasks, thereby substantially enhancing developer productivity.

Initially, code synthesis was grounded in formal logic theories, reflecting the early computational approaches that dominated the field. These foundational methods focused on deriving executable code from rigorous logical specifications (29; 23; 34). As datasets became larger and more accessible, the field of code synthesis began to incorporate machine learning techniques. This shift was facilitated by researchers like Husain (17), who explored how data-driven approaches could be employed to automate and enhance code generation processes. The introduction of deep learning brought a transformative change to code synthesis. Large-scale neural networks began to be utilized, enabling more sophisticated applications such as natural language processing to program synthesis. During this phase, notable models like CodeBERT, PyMT5 and Neural Sketch Learning for Conditional Program Generation were developed (10; 22; 15), utilizing vast code datasets to significantly improve programming capabilities, demonstrating the potential of AI models to handle complex programming functions effectively.

The landscape of automated code generation has been profoundly reshaped with the advent of advanced AI models, marking a new era in software development. Notably, the release of OpenAI's ChatGPT in November 2022, which rapidly attracted over 100 million users within two months, epitomizes this significant shift towards utilizing large language models (LLMs) for diverse applications, including coding. Further exploration (7) into GPT-3 revealed its remarkable capability to generate not only coherent text but also simple code snippets, demonstrating substantial generalization potential even without targeted training in code generation. This discovery spurred OpenAI to further innovate with (9), a derivative model specifically tuned on a sanitized dataset of Python code, alongside introducing new methodologies like the humanEval dataset and innovative sampling techniques, which propelled the era of large language models (LLMs) for coding . These advancements have paved the way for LLMs to undertake more complex programming tasks, pushing the boundaries of what automated systems can achieve. Additionally, recent advancements by DeepMind with models like AlphaCode have notably enhanced the capabilities of AI in programming by not only defeating 54.3% of human participants in coding competitions but also by extending the scope of automated code generation to competitive levels.

The development and effectiveness of LLMs for code generation are significantly hampered by two intertwined challenges: the limited quantity and questionable quality of available code data. Specifically, the data sourced from platforms like GitHub, while extensive, is finite and grows incrementally, which may not sufficiently represent the diversity of programming scenarios required for robust AI training. More critically, this data often lacks a systematic evaluation of code quality. The absence of quality metrics means that models are trained on datasets without discernment between high-quality, functional code and suboptimal or erroneous examples. This scenario can lead to the generation of flawed or inefficient code by the AI models.

To address these issues, a dual approach is proposed: enhancing the dataset both in size and quality. This involves not only expanding the quantity of code data available for model training but also implementing mechanisms to evaluate and categorize the quality of the code. By integrating tools that can assess code quality—potentially through automated testing and code features quality into the training process—models can be trained to discern and preferentially learn from higher-quality code. Additionally, leveraging suboptimal code by teaching models to recognize and correct errors could further enhance their practical utility and robustness. This approach promises to refine the training process for LLMs in code generation, ensuring they are both more capable and reliable in practical software development environments.

Chapter 2

Background Literature

Program synthesis is a long-standing research topic in computer science, having evolved over several decades since its inception. Early research in program synthesis dates back to the 1960s and 1970s, when researchers primarily focused on using logical reasoning and formal methods to automatically generate programs. However, these methods were limited in their applicability due to computational resource constraints and algorithmic complexity. As computational power and algorithms improved, program synthesis technology gradually expanded to more practical applications.

In the 21st century, particularly in recent years, the rise of deep learning technologies has brought new breakthroughs in program synthesis. Large language models (LLMs) based on pretrained transformers have achieved significant progress in program synthesis. These models, pretrained on vast amounts of code data, have demonstrated the ability to generate high-quality code. OpenAI's GPT series, particularly Codex (9), not only understand programming requirements described in natural language but also generate code snippets that match user intentions, significantly enhancing programmer productivity. Similarly, open-source multi-billion parameter code models like StarCoder (19) and Code LLaMA (28) have performed exceptionally well in code processing tasks, giving rise to commercial products such as GitHub Copilot.

The development of code processing models has seen a historical transition from statistical models and recurrent neural networks (RNNs) to pretrained transformers and large language models, mirroring the development trajectory in the field of natural language processing (NLP). While there have been attempts to integrate code-specific features such as abstract syntax trees (AST), control flow graphs (CFG), and unit tests into the training of language models, these approaches have not been the primary focus of most research efforts.

In recent years, reinforcement learning (RL) techniques have also been introduced into the field of program synthesis, enhancing the code generation capabilities of large language models through fine-tuning. Reinforcement learning guides models to generate code that better meets expectations through reward mechanisms, thereby achieving higher accuracy and reliability in complex programming tasks. This hybrid approach, combining deep learning and reinforcement learning, has opened new pathways for the future development of program synthesis technology, driving continuous advancements in the field.

Overall, program synthesis technology has evolved from early theoretical research to practical tools widely used in software development, data analysis, and other fields. As technology continues to advance, program synthesis is expected to further expand its application scope, enhancing the efficiency and accuracy of automated programming.

2.1 Pre-AI and AI Winter Period

Before deep learning and machine learning techniques were applied to program synthesis problems, early attempts included the use of heuristic methods such as the "General Problem Solver" (GPS). GPS originated from a project called "Logic Theorist," which focused on discovering proofs for sentential calculus theorems. Unlike the Logic Theorist, GPS addressed a broader range of problems and topics. GPS was implemented using a programming language called Information Processing Language (IPL), and it explored the complexity, creativity, and intelligence behind human problem-solving. Its primary goal was not to provide optimal solutions but to generalize as much as possible the recursive approach used by college students in discovering proofs. The heuristics used by GPS were similar to what is now known as dynamic programming, employing the "principle of subgoals reduction" to break down main goals into more manageable subgoals (23). One of the heuristics used by GPS was means-ends analysis. First, GPS identified the type of problem, whether it was a proof, an application of an operator, or an optimization problem. It then checked whether the problem was worthwhile to solve; if so, a method was selected and executed. Finally, the solution was evaluated, and these steps were recursively repeated until a solution was reached or the problem was deemed unsolvable according to the program(23). Following this, Simon conducted a series of experiments in 1963 on heuristic compilers built using a newer version of IPL called IPL-V, drawing on some problem-solving strategies from GPS (29). These early efforts laid the foundation for understanding program synthesis and paved the way for more accurate and advanced techniques.

In 1969, Waldinger and Lee published a detailed paper on the "PROW" project, which aimed to automate the writing of programs. PROW was constructed from a set of algorithms, primarily consisting of a theorem solver and a code generator. The theorem solver determined the algorithm's specification based on input in the form of predicate calculus, and the code generator then produced a LISP program according to the specification provided by the theorem solver.

The code generator could be adjusted to support any programming language. PROW's goal was to simplify the programming task, making it more enjoyable and allowing programmers to focus on higher-level problem-solving without needing to know specific or multiple programming languages (34). The paper also suggested that PROW might perform better with natural language inputs compared to predicate calculus inputs (34). This marked a shift from the logic-to-code approach to a text-to-code approach. However, before this transition, the performance of the theorem solver needed improvement. A few years later, in 1971, Manna and Waldinger further improved PROW's architecture, particularly in generating recursive and looping programs (21). This work demonstrated the use of mathematical induction principles to create recursive and looping programs, thereby improving the efficiency of the theorem solver.

In 1994, Anuchitanukul and Manna explored the realizability of concurrent programs, proposing a new approach that not only focused on the program synthesis algorithm but also introduced a realizability-checking algorithm to evaluate the feasibility of generating a program before attempting to create it (2). This method used a propositional Extended Temporal Logic (ETL)-based reactive module, differing from the predicate calculus used in the PROW project. This pre-generation evaluation technique was also employed in GPS (29), but in this paper, pre-generation evaluation was given greater importance as a standalone algorithm. The realizability-checking method assessed the feasibility of implementation given the ETL, building a structure for it if feasible, while the program synthesis algorithm generated the program by producing a labeled finite automaton (2).

In 1983, Wolper studied the expressive power of temporal logic, discovering that Extended Temporal Logic (ETL) was more expressive than Propositional Temporal Logic (PTL). This discovery had significant implications for program synthesis, as temporal logic could be used to precisely describe and reason about the temporal behavior of programs, providing a new tool to support automated program generation (39).

The AI Winter refers to periods from the 1960s to the late 1990s when funding for AI research significantly decreased. During this time, negative perceptions about AI capabilities were prevalent in the media, leading to a freeze in AI research (32). Many researchers believed that computers could not solve problems independently without human-provided code. For instance, Penrose in 1989 argued that AI behavior did not follow any algorithms and, therefore, could not be represented by a Turing machine (25). This misconception was later corrected with the rise of machine learning methods. Wang in 2007 and 2013 pointed out that AI models should not be directly associated with Turing machines, as models do not have fixed initial and final states (35; 36).

2.2 The Rise of AI Period

After the AI winter, the resurgence of artificial intelligence (AI) research brought significant advancements in program synthesis, driven by the introduction of artificial neural networks, transformers, and machine learning techniques. This period marked a transformation in natural language processing (NLP) and computer vision fields, with several key innovations and models shaping the future of program synthesis.

A notable advancement during this period was the introduction of the Non-Axiomatic Reasoning System (NARS) by Pei Wang. NARS utilized non-axiomatic logic and truth maintenance systems for problem-solving. This model was designed to handle incomplete and uncertain information, which traditional logic systems struggled with. NARS provided a robust framework for intelligent reasoning, allowing for more flexible and adaptive problem-solving capabilities. This approach demonstrated the potential of integrating logical reasoning with machine learning to enhance program synthesis (36).

The era of transformers began in 2017 with the seminal paper “Attention is All You Need” by Vaswani et al. This paper introduced the transformer architecture, which revolutionized NLP by enabling more efficient parallelization and improving performance on various NLP tasks. The key innovation was the attention mechanism, which allowed models to focus on different parts of the input sequence selectively. This capability significantly improved the handling of long-range dependencies in text, a critical aspect of program synthesis (33).

Following the introduction of transformers, OpenAI developed the Generative Pre-trained Transformer (GPT) models, progressively increasing in size and capabilities. GPT-3, released in 2020, was a significant milestone, showcasing the potential of large language models (LLMs) in text generation and program synthesis. GPT-3’s architecture, with 175 billion parameters, enabled it to perform a wide range of tasks with minimal fine-tuning, demonstrating remarkable versatility and capability in understanding and generating human-like text (7). GPT-3’s success in natural language understanding and generation opened new avenues for program synthesis. Its ability to generate code snippets, explain code, and even debug programs highlighted the practical applications of LLMs in software development. This development marked a shift towards using LLMs as powerful tools for automating and assisting in the coding process.

Building on the success of GPT-3, OpenAI introduced Codex (9), an AI model fine-tuned specifically for programming tasks. Codex powers applications like GitHub Copilot, which provides real-time code suggestions and completions based on the context of the written code. Codex was trained on a diverse dataset of natural language and code, enabling it to understand and generate code across various programming languages. This model exemplified the potential of fine-tuning LLMs on task-specific datasets to enhance their performance in specialized domains. Another important contribution is the introduction of the HumanEval benchmark, a set of tests specifically designed for evaluating code generation models.HumanEval consists of a variety of programming tasks that require the model to generate correct code that passes a series of test cases.

Following Codex, significant advancements in AI models for programming tasks have emerged, Zhang (40) have systematically review the recent advancements in code processing with language models, covering 50+ models, 30+ evaluation tasks, and 500 related works. These models range from general language models like the GPT family to specialized models pretrained specifically on code, often with tailored objectives. AlphaCode (20) represents a leap forward through innovations in model architecture, training data, and evaluation methodologies. AlphaCode utilizes a larger and more diverse training dataset, including coding problems and solutions from competitive programming platforms. This enables it to generalize across various programming tasks more effectively. Its transformer-based architecture is optimized for coding, enhancing performance and efficiency by better handling long-range dependencies and integrating domain-specific knowledge. A key innovation in AlphaCode is its rigorous evaluation protocol, testing the model on complex coding problems under competitive conditions. This approach ensures that the model's performance metrics reflect practical utility in real-world programming scenarios. These advancements highlight the importance of refining model architectures, expanding training datasets, and developing comprehensive testing frameworks.

Meanwhile, various optimization approaches and hybrid methods emerged, combining traditional AI techniques with modern machine learning models. These methods aimed to leverage the strengths of both paradigms to improve program synthesis. For instance, integrating symbolic reasoning with neural networks allowed for more interpretable and logically consistent outputs while maintaining the flexibility and learning capabilities of neural networks. One notable example is the development of retrieval-augmented models like CodeBERT (15), which combined pre-trained language models with retrieval mechanisms to enhance code completion and generation tasks. CodeBERT leveraged the contextual understanding of pre-trained models and the precision of retrieval systems to provide more accurate and relevant code suggestions (15). This hybrid approach illustrated the benefits of combining different AI techniques to address the complexities of program synthesis effectively.

In the field of sequence generation, Reinforcement Learning (RL) methods have achieved significant success by leveraging non-differentiable task-specific metrics. For example, Ranzato (27) used the REINFORCE algorithm to directly optimize translation models for BLEU and ROUGE scores. Similarly, Bahdanau (4) introduced an actor-critic framework to enhance sequence generation tasks. More relevant to our work are RL-based program synthesis methods. Bunel (8) explored RL for program generation, while Ellis (13) investigated execution-guided synthesis methods. However, these methods are often limited to specific programming languages and application domains.

Austin (3) and Chen (9) emphasized that in program synthesis, traditional token-based similarity metrics have a low correlation with functional correctness. To address this issue, Le (18) studied the integration of RL with unit test signals in the fine-tuning of program synthesis models, and proposed codeRL model. They proposed leveraging unit test signals during both model optimization and test-time generation stages, as unit test signals directly reflect the functional correctness of programs. However, in RL, the characteristics of code have not been used as a reward signal, unlike many studies in transformer-based code language modeling that have incorporated these features into their training processes.

As outlined in previous discussions, the landscape of code generation systems encompasses a variety of methodologies, each contributing uniquely to the field. Central to this project is a training and evaluation strategy inspired by the (18), renowned for its effectiveness in current applications. Furthermore, it has been noted that while codeRL leverages the complexity and pass rates of competitive programming solutions as metrics for evaluation, it lacks a scoring system that accounts for intrinsic code qualities. To address this gap, this project proposes the integration of underlying code characteristics, such as data flow and syntactic-semantics, to enhance model performance in targeted use cases. This approach aims not only to refine the evaluation metrics but also to enrich the model's understanding and generation of code, thereby facilitating a more robust.

Chapter 3

Lyrics Generation - Methodology

In the preceding chapters, we have not yet discussed the application of LSTM-based models in the task of lyric generation, as I intend to dedicate an entire forthcoming chapter to the design process of this task. Although this model initially demonstrated its capability for generating lyrics, further exploration of its potential and limitations revealed critical bottlenecks in the project. Specifically, the project faced issues such as the lack of established criteria for assessing lyric quality and inherent flaws within the model itself. Furthermore, upon reviewing recent developments in large language models, it became apparent that pretrained large models could be efficiently fine-tuned to enhance performance on specific tasks. Consequently, this project shifted focus towards employing a large model based on reinforcement learning for the task of code generation, a method which recent studies have shown to possess higher efficiency and flexibility.

3.1 Lyrics generation task based on LSTM

My original topic was lyrics generation. In the lyrics generation task, a sequence-to-sequence (Seq2Seq) approach can be described as a supervised learning problem in which the model learns a mapping from an input sequence (e.g., a series of words, phrases, or previous lyrics) to an output sequence (the next lyric). The key here is to consider the lyrics generation task as a sequence prediction problem, where the model needs to predict the next sequence of lyrics, based on given historical lyrics data.

1. **Input and Output Sequences**:

    - Input sequence $D = (d_1, d_2, \ldots, d_S)$ represents a sequence describing the problem, such as a set of trigger words or initial lines of lyrics.

- Output sequence $\hat{W} = (\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_T)$, where each $\hat{w}_t \in V$, is the sequence of generated lyrics, with each $\hat{w}_t$ being an element from the vocabulary V.

2. **Model Architecture**:

   - **LSTM Encoder**: Transforms the input sequence D into a context vector C, which captures the essential information of the input.

   $$C = f_{\text{LSTM\_encoder}}(D)$$

   - **LSTM Decoder**: At each decoding step t, uses the context vector C and the outputs generated so far $\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_{t-1}$ to predict the next output $\hat{w}_t$.

   $$\hat{w}_t \sim \text{softmax}(\text{Linear}(s_t))$$

   where s_t is the hidden state at decoding step t in the LSTM decoder.

3. **Learning Objective**:

   - During the training phase, model parameters θ are learned by maximizing the likelihood of the ground-truth reference lyrics. Assuming $W = (w_1, \ldots, w_T)$ is the ground-truth lyrics sequence, the objective is to minimize the cross-entropy loss:

   $$L_{\text{ce}}(\theta) = -\sum_t \log p_\theta(W|D) = -\sum_t \log[p_\theta(w_t|w_1 : t-1, D)]$$

   where the conditional probability p_θ is parameterized by the aforementioned softmax function.

4. **Decoding and Generation**:

   - During test time, the model generates sequences of lyrics by autoregressively sampling tokens $\hat{w}_t$ from the vocabulary V. This process relies on previous outputs and the context state to adaptively generate the next most appropriate token.

DNN(Deep Neural Networks) can solve a lot of problems, but its biggest problem is that it can't consider the time series problems, in addition, DNN requires that the dimension of each batch input is the same, which is difficult to predict in advance in the process of machine translation, so there is a seq2seq model. seq2seq is essentially an encoder-decoder model. Specifically on our lyrics generation task, in the lyrics generation task, the input can be certain keywords, sentiment tags, or existing lyrics from previous lines. The role of the encoder is to accept the input sequence (e.g., a piece of cue text or the first few lines of lyrics) and convert it into a fixed-length internal representation or context vector c. This context vector contains key information about the input sequence and is used to generate the output sequence. The decoder's task is to generate the output sequence based on the context vector c provided by the encoder. In the scenario of lyrics generation, this usually means generating one or several lines of lyrics. It generates the output sequence step by step, producing a new word at each step. While generating each word, the decoder takes into account the word generated in the previous step and the encoder's context vector, thus ensuring that the generated lyrics are semantically compatible with the input. DNN(Deep learning model) can solve a lot of problems, but its biggest problem is that it can't consider the time series problems, in addition, DNN requires that the dimension of each batch input is the same, which is difficult to predict in advance in the process of machine translation, so there is a seq2seq model. seq2seq is essentially an encoder-decoder model. Specifically on our lyrics generation task, in the lyrics generation task, the input can be certain keywords, sentiment tags, or existing lyrics from previous lines. The role of the encoder is to accept the input sequence (e.g., a piece of cue text or the first few lines of lyrics) and convert it into a fixed-length internal representation or context vector c. This context vector contains key information about the input sequence and is used to generate the output sequence. The decoder's task is to generate the output sequence based on the context vector c provided by the encoder. In the scenario of lyrics generation, this usually means generating one or several lines of lyrics. It generates the output sequence step by step, producing a new word at each step. While generating each word, the decoder takes into account the word generated in the previous step and the encoder's context vector, thus ensuring that the generated lyrics are semantically compatible with the input.

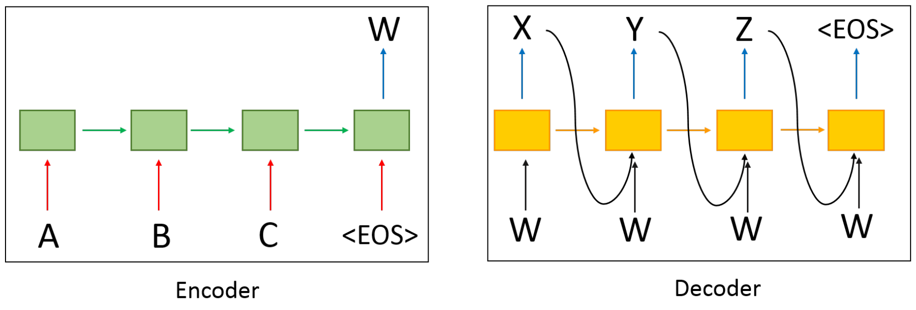


Figure 3.1: Illustration of the Model Construction

Step 1: Read Training File and Tokenization

The lyric data is pulled directly from a Kaggle repository, categorized by different artists, and includes a multilingual dataset in both English and Chinese. For tokenization, NLTK tokenizer is used considering its adaptability to multiple languages. The purpose of indexing is to facilitate the input of sequences in the network model for data representation.

Step 2: Training and Test Sets

In the Seq2Seq model, the input is a sentence, and the output is also a sentence. Sample construction is as follows:

- Lyric sentences:
- ”By myself sometimes”
- ”To give my mind some space”
- ”Yeah I know, yeah I know that it hurts”
- ”When I push your love away, I hate myself”

Generated data example (X represents input data, y represents output labels):

- x1: ”By myself sometimes”, y1: ”To give my mind some space”
- x2: ”To give my mind some space”, y2: ”Yeah I know, yeah I know that it hurts”
- x3: ”Yeah I know, yeah I know that it hurts”, y3: ”When I push your love away, I hate myself”

Step 4: Model Construction

Both the decoder and encoder parts are based on LSTM, there is a dense layer after the decoder and after that a softmax layer is connected for prediction.

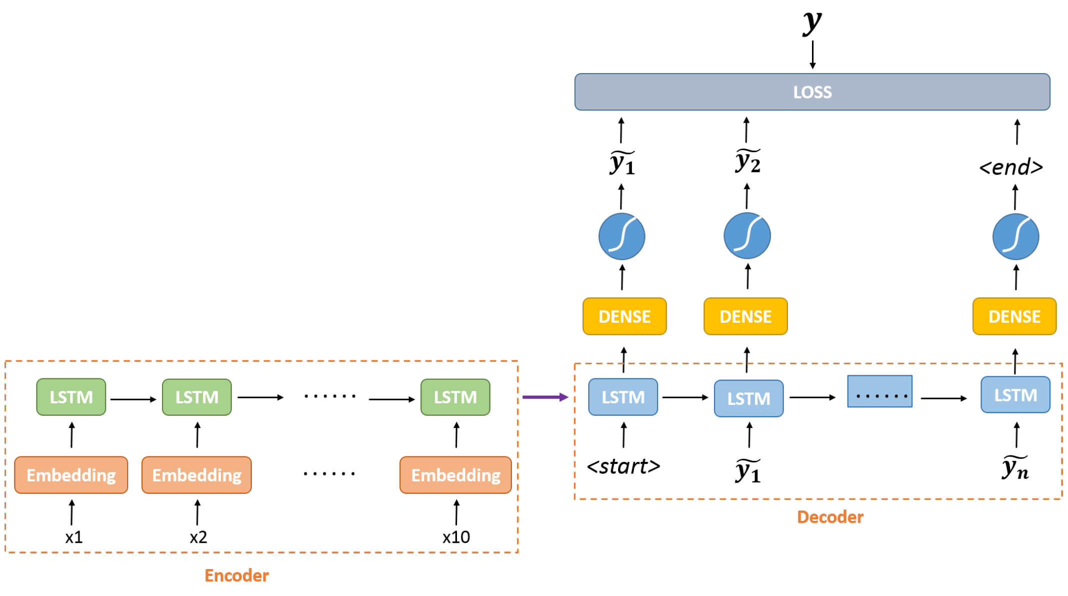


Figure 3.2: Illustration of the Model Construction

Step 5: Parameter Tuning

- When the learning rate is set to 0.001, val_acc starts to decline from the 10th epoch, indicating early overfitting. Acc accuracy initially rises quickly and then stabilizes. Val_loss begins to rise after a certain period, which is also a sign of overfitting, indicating good model performance on training data (low loss) but poor performance on new data (increased loss). The loss still follows a normal declining trend. This indicates that the learning rate is too high and training does not reach the optimal state. Adjusting the learning rate to 0.0001 solves the problem.
- LSTM layers: Changing from 1 layer of LSTM to 2 layers, the result with 2 layers is better than that with 1 layer. The best result with 2 layers LSTM is val-loss $\approx$ 3.31, val-acc $\approx$ 0.55; with 1 layer LSTM, the best result is val-loss $\approx$ 4.34, val-acc $\approx$ 0.43.
- Optimization algorithm: Changing from Adam optimization algorithm to RMSprop algorithm, results show that the Adam algorithm performs better, specifically reflected in smaller val_loss and higher val_acc.
- Data: Initially using data from one artist resulted in very poor outcomes, expanding the dataset significantly improved the results. To further improve the results, consider continuing to expand the training set.

Step 6: Data Accuracy Analysis

Observing the two loss graphs on the right, before the 15th epoch, the loss on the training set continuously decreases and the val-loss on the validation set also continuously decreases, indicating that the model is continuously fitting the data. However, after the 15th epoch, the loss on the training set continues to decrease while the val-loss on the validation set begins to continuously increase, indicating typical overfitting issues. One question is why in the validation set, val-loss continuously increases after 15 epochs, but val-accuracy continues to rise. Possible reasons are, first, after 15 epochs, although val-accuracy also continues to increase, its trend is very stable, meaning that although val-loss has increased, as long as the prediction score does not exceed the critical value of category change, it can still maintain a relatively stable state even when val-loss increases. Second, in situations where the data volume is not very large, the aforementioned issue indeed occurs, but if the data volume is sufficiently large, the eventual stable situation is that as loss increases, acc will decrease.

In summary, in training network measurement indicators, loss is used more commonly than accuracy because: (1) the loss function is differentiable, whereas accuracy is not differentiable, and a differentiable objective function is needed in the network model backpropagation process. (2) In classification tasks, using accuracy is possible, but in regression tasks, accuracy is no longer applicable, only loss can be used. (3) There are multiple optimization methods for loss functions, such as Newton's method, maximum likelihood estimation.

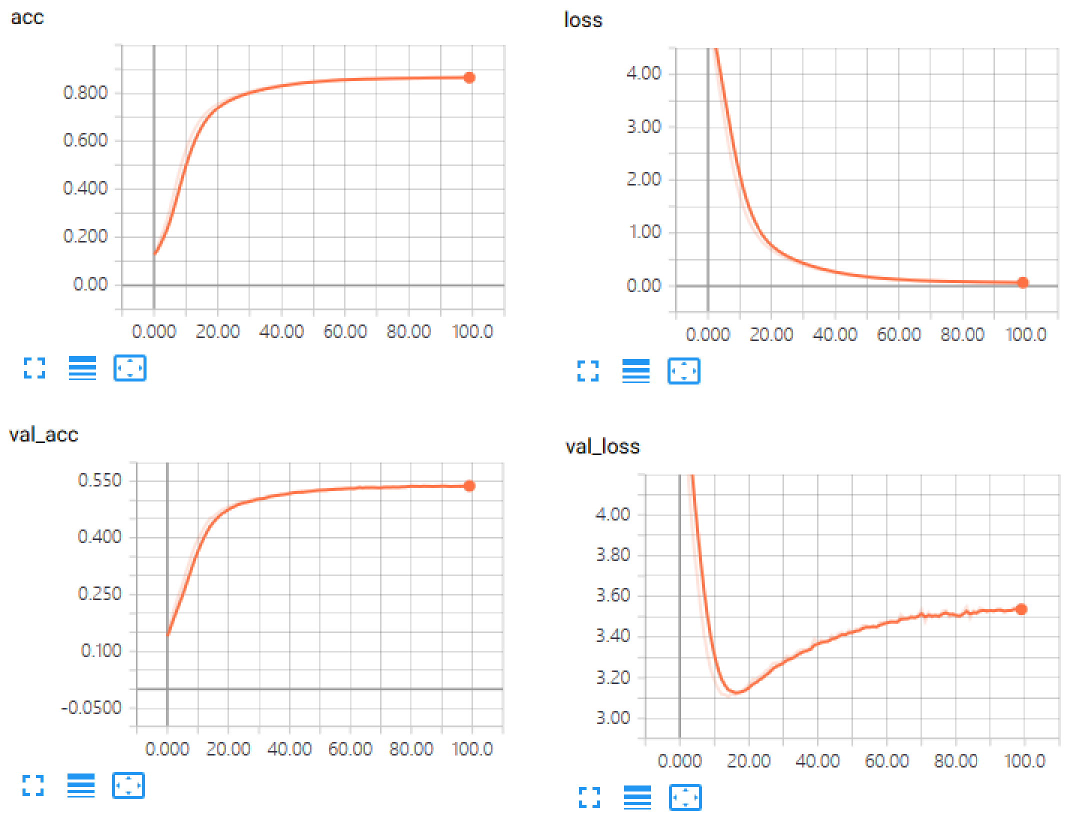


Figure 3.3: Illustration of the Model Construction

Future Work

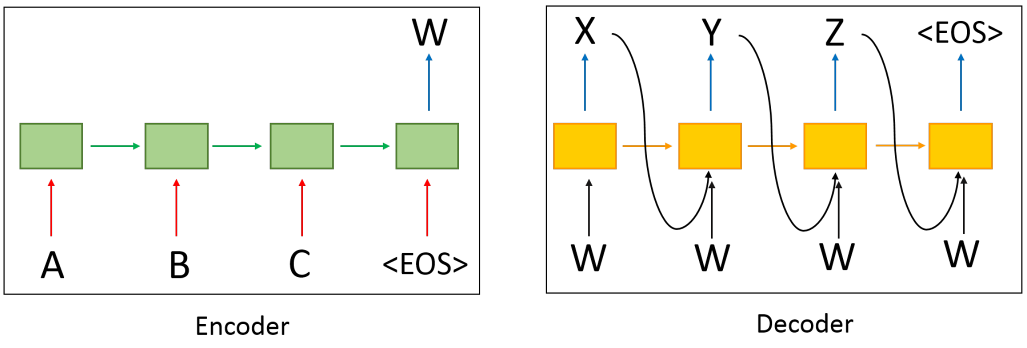


Figure 3.4: Basic Seq2Seq Model - Disadvantage

In the decoding process of a Seq2Seq model, the semantic vector W, generated by the encoder, is initially fed into the model to produce X; due to the interdependence of the sequence, not only is W inputted but also the previously generated X to produce Y; subsequently, both W and Y are used to generate Z. However, an issue can be discerned from this structure. Firstly, as observed in the diagram, the same W is used throughout the decoding process, but for lyric generation tasks, each input word has a different influence on the current output. For instance, if a verse begins with ”heartbreak”, the subsequent lines should ideally focus on emotions related to ”sorrow” or ”loneliness” rather than shifting to unrelated themes. Secondly,

W is required to store information from all previous words, and the length of the sequence becomes a significant bottleneck. A single, fixed-length semantic vector W may struggle to capture all necessary information, leading to loss of information, especially in longer sentences. For example, if the first verse sets a scene, subsequent paragraphs need to develop based on this setting. Traditional Seq2Seq models may find it challenging to handle long contextual sequences, potentially leading to generated lyrics that do not align with the overall theme and emotions of the song.

The better way lies in incorporating an attention mechanism using an Attention-based Seq2Seq Model. The principle behind this mechanism is that it allows the model to ”focus” on different parts of the input sentence while generating each word. This means that the model does not solely rely on a fixed semantic vector W; instead, it dynamically extracts information from the entire input sequence based on the content currently being generated.

Shift to Code Generation

The problem of judging criteria:

The evaluation criteria of lyrics generation are often subjective, because the quality of lyrics depends largely on individual aesthetics and emotional resonance, which makes it difficult to optimize the model and evaluate its performance. For example, although the generated lyrics can be evaluated by some technical indicators (e.g., lexical richness, rhyming patterns, etc.), these indicators can hardly fully reflect the artistic value of the lyrics and the actual feelings of the listeners, meanwhile there is a lack of benchmarks for this task. On the contrary, the task of code generation has more clear and objective evaluation criteria, such as the functionality, correctness and efficiency of the code, which makes the training and evaluation process of the model more clear and controllable.

Limitations of the model’s capabilities:

In the practical application of LSTM for lyrics generation, I found that while LSTM was able to handle short to medium length sequence dependencies, its effectiveness gradually declined when dealing with long sequences. In particular, when trying to extend the model to handle more complex generative tasks (e.g., maintaining coherence and thematic consistency of long lyrics), more data and deeper network hierarchies are often required to achieve this. This not only leads to a significant increase in training costs, but also makes deployment and maintenance of the model more difficult. In contrast, the latest large-scale language models, such as the GPT family, adopt the Transformer architecture, which efficiently handles long-distance dependencies through the self-attention mechanism and has better scalability and parallel processing capabilities. In addition, the large-scale language models have learned rich language representations by pre-training on a wide range of texts, like large amounts of complex linguistic, emotional and cultural contextual information, and can be adapted to different downstream tasks, including code generation, more efficiently.

Chapter 4

Code Generation - Methodology

A significant difference between programming languages and natural languages lies in their precision and unambiguity. Programming languages are designed to be precise and clear, with strict syntax and semantics, ensuring that code can be compiled or interpreted without errors before execution. This precision eliminates much of the ambiguity inherent in natural languages, making the evaluation of generated code more straightforward and objective compared to evaluating the artistic and subjective elements of lyrics.

While large language models (LLMs) fine-tuned on code corpora have achieved success in program synthesis, they often overlook essential signals such as underlying code features. Our project addresses this by incorporating code-specific characteristics like Abstract Syntax Tree (AST), Control Flow, Data Flow, and Type information into the code generation process. These elements provide a structural and semantic context that is crucial for generating high-quality code.

During training, we treat the code-generating LLM as an actor network and introduce a critic network. The critic network is trained to predict the functional correctness, compilability, and overall quality of generated programs, providing feedback to the actor. This feedback loop ensures that the generated code not only syntactically correct but also functionally robust.

Additionally, we propose a pipeline for generating high-quality code data. This pipeline leverages feedback from competition tests and feature checks to continually improve the training data. Our method achieves state-of-the-art results on the challenging APPS benchmark and the simpler MBPP benchmark, demonstrating robust zero-shot transfer capabilities, where the model effectively generalizes to new, unseen tasks without additional training.

4.1 Program Synthesis Task

In the previous chapter, I have explained how the task shifted from lyrics generation to code generation. The task of code generation is also a sequence-to-sequence generation task. In the specific context of coding competitions, the input sequence is a description D of a problem, while the ideal output sequence is a correct program $W = (w_1, w_2, \dots, w_T)$ that can run, correctly compile, and pass all the tests. The code generation process involves multiple components each performing a distinct function within the sequence-to-sequence model framework.

Initially, the encoder transforms the input sequence $D = (d_1, \dots, d_N)$ into a sequence of context vectors $C = (c_1, \dots, c_N)$. Each vector c_i encapsulates a compressed representation of the input data. This encoding is formalized as $c_i = f_{\text{enc}}(d_i, h_{i-1}^{\text{enc}})$, where f_{enc} represents the encoder's recursive function and h_{i-1}^{enc} denotes the preceding hidden state.

The decoder sequentially generates the output sequence $\hat{W} = (\hat{w}_1, \dots, \hat{w}_T)$, where each token $\hat{w}_t$ is derived as follows: $\hat{w}_t \sim \text{softmax}(\text{Linear}(s_t))$ and $s_t = f_{\text{dec}}(s_{t-1}, \hat{w}_{t-1}, c_t)$. In this formulation, f_{dec} signifies the decoder's recursive function, s_t is the current hidden state, and Linear refers to a linear transformation layer.

The attention mechanism enables the decoder to selectively focus on different segments of the input sequence through the computation of attention weights: $a_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{N} \exp(e_{t,j})}$, $e_{t,i} = \text{align}(s_{t-1}, c_i)$, and $c_t = \sum_{i=1}^{N} a_{t,i} c_i$. Here, align denotes the alignment model, typically implemented via a feed-forward network or a scalar dot product.

The objective function, aimed at minimizing the divergence between predicted and actual programs, is represented by the cross-entropy loss: $L_{\text{ce}}(\theta) = -\sum_t \log p_\theta(w_t \mid w_{1:t-1}, D)$. This function quantifies the divergence between the predicted output and the actual program, guiding the optimization of the model parameters θ through algorithms like Stochastic Gradient Descent (SGD) or Adam.

During test time, the models generate sequences of programs by autoregressively sampling tokens $\hat{w}_t$ from the distribution $p_\theta(.|\hat{w}_{1:t-1}, D)$. This can be implemented using strategies such as greedy decoding or beam search. The generated sequences are then evaluated against unit tests corresponding to the problem, where each test includes a pair of input and ground-truth output. In real-world program synthesis tasks, example unit tests are often given as parts of the problem specification.

4.2 Reinforcement Learning Framework

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. In the context of fine-tuning large language models (LLMs), RL can be used to optimize the model's performance by continuously refining its outputs based on various scoring mechanisms. RL is particularly suitable for tasks such as code generation, where the model must produce syntactically and semantically correct code. By leveraging RL, we can iteratively improve the model's ability to generate high-quality code by incorporating feedback from compilers, syntax checkers, and semantic analyzers. The primary reason for choosing RL for code generation tasks is its ability to handle sequential decision-making problems. Code generation can be viewed as a sequence of actions (token generation), where each action impacts subsequent ones. RL allows the model to learn from the entire sequence rather than just individual steps, thereby improving the overall quality of the generated code.

A simplified Markov Decision Process (MDP) can be represented as a quadruple (S, A, P, R), where:

State Space (S) represents all possible states of the system. In our case, a state s_t at time step t is represented by the current partially generated program and the input problem description x. Formally,

$$S_t = (Y_{<t}, X),$$

where $Y_{<t}$ is the sequence of tokens generated up to time t. In the context of program synthesis, At each time step t, the state s_t includes the sequence of tokens generated so far $Y_{<t}$ and the input problem description x.

Action Space (A) represents all possible actions the agent can take in each state. An action a_t corresponds to the selection of a specific token w_t at time step t. In the context of program synthesis, the action a_t is the generation of the next token W_t.

State Transition Probability (P) represents the probability of transitioning to state s' after taking action a in state s. This is determined by the policy network, which generates the next token based on the current state.

Reward Function (R) represents the immediate reward received after taking action a in state s. This reward evaluates the quality of the generated token in terms of its contribution to generating a syntactically and functionally correct program.

A policy network, denoted as π, is a strategy used by an agent to decide on an action based on the current state. Formally, the policy $\pi_\theta(a_t|s_t)$ is a probability distribution over the action space A given the current state s_t, where θ represents the parameters of the policy network. In our approach, we use a pre-trained language model, such as CodeT5, as the policy network. This model predicts the next token w_t given the current state s_t. The learned parameters θ of the language model represent a stochastic policy, meaning that the model decides the action (i.e., the next token) probabilistically based on the current state. After each action, the model updates its hidden state representations, which are then used to determine the next action in the subsequent decoding step. At the end of the generation episode (i.e., when an `<endoftext>` token is generated), the policy receives a return r based on the functional correctness of the generated program. The objective of reinforcement learning fine-tuning is to maximize the expected return by adjusting the parameters θ to improve the quality of the generated programs.

The goal of reinforcement learning (RL) in our context is to maximize the expected return from the generated programs. Formally, the expected return $J(\theta)$ is defined as:

$$J(\theta) = E_{W^s \sim \pi_\theta}[r(W^s)] \tag{4.1}$$

where $W^s = (w_1^s, \ldots, w_T^s)$ represents a sequence of tokens generated by the language model under policy π_θ. The return $r(W^s)$ is a function that measures the functional correctness and other quality metrics of the generated program W^s. To facilitate optimization, we define the RL loss function as the negative expected return:

$$L_{rl}(\theta) = -J(\theta) = -E_{W^s \sim \pi_\theta}[r(W^s)] \tag{4.2}$$

Using the REINFORCE algorithm (38) and policy gradient theorem (31), the gradient of the non-differentiable return r with respect to the policy parameters θ can be estimated as:

$$\nabla_\theta L_{rl}(\theta) \approx -E_{W^s \sim \pi_\theta}[r(W^s) \nabla_\theta \log p_\theta(W^s|D)] \tag{4.3}$$

This equation shows that the gradient of the RL loss function can be approximated by taking the expectation over the product of the return $r(W^s)$ and the gradient of the log-probability of the generated sequence $\log p_\theta(W^s|D)$.

The gradient can further be broken down into token-level contributions:

$$\nabla_\theta L_{rl}(\theta) \approx E_s[r(W^s) \sum_{t=1}^{T} \nabla_\theta \log p_\theta(w_t^s|w_{<t}^s, D)] \tag{4.4}$$

Here, the gradient is expressed as the sum of the gradients of the log-probabilities of each token w_t^s in the sequence, weighted by the return $r(W^s)$. To optimize the policy parameters θ, we use gradient descent methods to update the parameters:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} L_{rl}(\theta) \tag{4.5}$$

where α is the learning rate. By iteratively applying these steps, the policy (LM parameters θ) is fine-tuned to maximize the expected return, leading to the generation of more functionally correct programs.

4.3 Model

This project uses the CodeT5 model to fine-tune a large-scale language model through reinforcement learning for competition-level code generation tasks with only one 16GB V100 GPU. CodeT5, proposed by Wang(37) a multilingual code-aware language model pretrained on a large-scale GitHub code corpus, demonstrating state-of-the-art performance in code understanding and generation tasks.

CodeT5 is a unified encoder-decoder model based on the Transformer architecture, specifically designed for code understanding and generation tasks. The model has achieved state-of-the-art performance on the CodeXGLUE benchmark, supporting a variety of tasks including code summarization, code generation, and code translation. Its pretraining objectives include Masked Span Prediction (MSP), Identifier Tagging (IT), and Masked Identifier Prediction (MIP), which help the model better understand the semantics and structure of code. According to the experimental results in the literature, CodeT5 outperforms existing state-of-the-art models in multiple tasks. For instance, it achieves the highest BLEU-4 scores in tasks such as code summarization, code generation, and code translation, demonstrating its strong capabilities in code understanding and generation.

Reasons for Model Selection:CodeT5-small model has around 60M parameters, and the CodeT5-base model has 220M parameters, making them suitable for training and fine-tuning with limited hardware resources such as a 16GB V100 GPU. CodeT5 employs identifier-aware pretraining objectives that capture the semantic information of the code, improving performance on code generation and understanding tasks. Additionally, CodeT5 supports multi-task learning, enabling it to handle various code-related tasks within a single model, thereby enhancing its generalization capability and practicality.

Advantages of CodeT5 as a Critic

1. **Zero-Shot Transfer Capability**: CodeT5 has demonstrated strong zero-shot transfer capabilities on benchmarks like MBPP, achieving state-of-the-art results. This indicates its

robust ability to generalize from pretraining data to unseen tasks, making it an excellent critic for providing feedback on novel and complex coding problems encountered during reinforcement learning.

2. **Comprehensive Code Understanding**: The model is trained on a vast corpus of multilingual code from GitHub, allowing it to develop a deep understanding of code syntax and semantics across various programming languages. This extensive pretraining makes CodeT5 highly effective at identifying errors and predicting the functional correctness of code, which are essential tasks for a critic in an RL framework.

3. **Dense Feedback Signals**: In reinforcement learning framework, the critic network is responsible for providing dense feedback signals to the actor network. CodeT5, with its sophisticated understanding of code, can accurately assess the functional correctness of programs and offer detailed feedback, guiding the actor towards generating correct and optimized code solutions.

4.4 Data

Given that our task involves fine-tuning LLMs using reinforcement learning (RL) models, we harness the inherent code generation capabilities of LLMs. The primary advantage of this approach is that it allows us to create a tailored dataset without relying on external sources, thereby ensuring data relevance and quality. In this context, we utilized the APPS (Automated Programming Progress Standard) competition problems (16) as our primary dataset. These problems provide a robust benchmark for evaluating the code generation capabilities of our models. We specifically employed the APPS evaluation framework to assess the model's ability to solve coding challenges, which is a critical aspect of its performance. Furthermore, we incorporated compiler feedback, semantic match score, and syntactic match score as part of our preprocessing strategy to enhance the dataset quality. Compiler feedback helps in identifying and rectifying errors in the generated code, ensuring it is syntactically correct and executable. The semantic match score evaluates the correctness of the generated code in terms of its intended functionality, ensuring the solutions are logically accurate. The syntactic match score assesses the structural accuracy of the code, ensuring it adheres to the expected coding patterns and conventions.

4.4.1 Data collection

The goal is to establish a pipeline capable of generating a substantial number of high-quality answers for each coding problem. This will support the subsequent training of reinforcement learning models by providing a rich and varied dataset, thereby improving the model's practical utility and effectiveness in solving coding issues. To achieve this goal, we wrote a script and used the CodeT5 model for inference, aiming to generate as much data as possible. Due to time constraints, we generated 25 answers for each of the 5000 problems in the apps dataset. The inference process ran for 24 hours on a V100 GPU, primarily limited by time. The choice of V100 GPU was due to its performance and efficiency in handling large models, but even so, time constraints remained a critical factor. Currently, we are not filtering the generated code. In the next phase, we will label the code: those that fail to compile and do not pass tests will be given negative scores, while functionally correct code will receive positive scores. This approach is intended to automatically filter out low-quality code in subsequent stages, ensuring the high quality and reliability of the training data.

During the execution process, The ECS GPU computing service at the University of Southampton presented several challenges in this process. Initially, we used the Yann server with four GTX 1080Ti GPUs. However, the GTX 1080Ti does not support native FP16 (half-precision) computation and only supports FP32 (single-precision) computation. Modern large model inference and training widely use half-precision and mixed-precision training to improve computational efficiency and reduce memory usage. Additionally, the GTX 1080Ti's memory capacity of 11GB is clearly insufficient for the CodeT5 770M model. The inference process requires significant memory to store model weights, intermediate activations, and other temporary data.

Although optimization libraries like DeepSpeed can enhance model training and inference efficiency, and distributed training techniques can partition the model across multiple GPUs, these methods are still ineffective when constrained by memory and computational capacity. The memory limitations of the GTX 1080Ti hinder the practical benefits of DeepSpeed (1) optimization. Distributing the model across multiple GPUs necessitates efficient communication and data exchange between GPUs. The GTX 1080Ti's memory bandwidth and communication efficiency are relatively low, resulting in limited performance gains and increased complexity for distributed inference. More importantly, distributed computing introduces additional computational and communication overhead, further reducing inference performance and failing to address the fundamental issue of insufficient memory. Therefore, multi-GPU training and inference with the GTX 1080Ti is not an ideal choice.

The best option is to use modern GPUs such as the NVIDIA RTX 30 series or A100 for multi-GPU training and inference. However the HPC service at Southampton only allows me to use a single V100 GPU with 16GB of memory, but this is still a significant improvement. Although the inability to use multi-GPU parallelism is a limitation, using DeepSpeed to offload some memory to the CPU can still accelerate the inference and training processes.

4.4.2 Data Cleansing and Pre-processing

After the program is generated, it will be evaluated against unseen unit tests for each problem, based on the syntactic match score using the AST and the semantic match score using the DFG, scoring different aspects of the generated code.

In this scoring system, the generated code undergoes multiple steps and different types of evaluations to assess its quality. First, for each source data, multiple codes are sampled from the current policy network and passed to a compiler. The compiler returns a reward based on the parsing signal. For the APPS dataset, it provides a testing interface that can be directly used to perform unit tests on the generated programs. Specifically, the generated code runs a series of predefined test cases through the APPS interface. If the code passes all test cases, it is considered functionally correct and receives a positive score; if the code fails any test case, encounters a runtime error, or a compile error, it receives varying degrees of negative scores. Additionally, the code is evaluated based on the syntactic match score using the abstract syntax tree (AST) and the semantic match score using the data flow graph (DFG), scoring different aspects of the generated code. These combined scoring mechanisms ensure that the system can comprehensively evaluate the quality of the generated code, considering both syntactic structure and semantic logic, thus ensuring the high quality and correctness of the code.

Subsequently, the abstract syntax tree (AST) is used to compare the structure of the generated code and the target code, evaluating the syntactic match score by calculating the percentage of matched AST subtrees to the total number of target AST subtrees. Additionally, the data flow graph (DFG) represents the variable dependencies within the code, and the semantic match score is evaluated by calculating the percentage of matched DFG edges to the total number of target DFG edges. The combined scoring mechanisms ensure that the system can comprehensively evaluate the quality of the generated code, taking into account both syntactic structure and semantic logic, thereby ensuring high-quality and correctness of the code.

4.4.3 Data Analysis

We conducted data analysis on a total of 12,500 generated code samples. Among them, 1,047 had compiler errors, only 116 passed at least one test, and none passed all the tests. I believe the primary reason lies in the model itself. Although CodeT5-770M was pretrained on CodeSearchNet (17), it is still a model with insufficient parameters.

Why did we not analyze code diversity, code length and complexity, error coverage, and compile-time and run-time errors in detail? Because these analyses require extensive text comparisons of the generated code, which involves significant computational resources. Additionally, high code duplication may indeed indicate a lack of innovation in the model, but in the initial stage of our research, we are more concerned with the functional correctness and quality of the generated code rather than its diversity.

To elaborate:

1. **Code Duplication Rate**: Analyzing the duplication rate of the generated code requires extensive text comparison, which is computationally expensive.
2. **Code Length and Complexity**: Statistics on the length of the generated code, the number of functions, and complexity metrics (such as loop nesting depth and number of conditional branches) require detailed code structure analysis.
3. **Error Coverage**: Evaluating the performance of the generated code under different error conditions requires constructing a comprehensive set of error test cases and performing extensive error handling tests.

4.5 Enhanced Reward Function

For each source data x, we sample multiple generated codes in the target language based on the current policy network, $\hat{y} \sim \pi_\theta(\cdot|x)$. Then, we pass these sampled codes $\hat{y}$ to a compiler and determine the reward based on the parsing signal. In case unit tests are available for the source data, the reward is determined by the functional correctness of generated codes, i.e., passing all unit tests, as shown in Eq. (4.6). If unit tests are not provided, compiler returns the syntactic correctness of generated codes as shown in Eq. (4.7). This reward term is designed to provide guidance to the model, encouraging it to take actions that result in the generation of higher-quality code in terms of syntactic/functional correctness.

4.5.1 Compiler Signal

Compiler feedback indicates syntactic and functional correctness. Compiler feedback is obtained from the compiler's compilation results or unit test results of the generated code. This signal is intended to ensure that the generated code is syntactically and functionally correct. Successful compilation, runtime errors and passing of unit tests can be used as the basis for feedback. Compiler feedback provides a direct and clear signal as to whether the generated code can be understood and executed by the machine. This is a strong indicator because the code ultimately needs to be run in a real-world environment. Compiler feedback is usually discrete (pass or fail), and thus may lead to sparse reward signals, affecting the efficiency of model training. To mitigate this problem, fine-grained error information can be introduced, e.g., by assigning different penalty weights to compilation errors, runtime errors, and unit test failures, respectively.

Functional Correctness:

$$R_{cs}(\hat{y}) = \begin{cases} +1, & \text{if } \hat{y} \text{ passed all unit tests} \\ -0.3, & \text{if } \hat{y} \text{ failed any unit test, but still compiles} \\ -0.6, & \text{if } \hat{y} \text{ received RunTime error} \\ -1, & \text{if } \hat{y} \text{ received Compile error} \end{cases} \tag{4.6}$$

Syntactic Correctness:

$$R_{cs}(\hat{y}) = \begin{cases} +1, & \text{if } \hat{y} \text{ passed compilation test} \\ -1, & \text{otherwise} \end{cases} \tag{4.7}$$

4.5.2 Syntactic Matching Score

To enhance the guidance of the policy samples beyond the sparse compiler signals, we incorporate additional syntactic information. We define a syntactic matching score $R_{\text{ast}}(\hat{y}, y)$ which measures the similarity between the generated code $\hat{y} \sim \pi_\theta(\cdot|x)$ and the reference target code y. This score is computed by comparing the abstract syntax trees (ASTs) of the generated and target code, quantifying the percentage of matched AST sub-trees. This method identifies structural errors such as missing operators or incorrect type conversions, which are often missed by simple textual comparisons. By examining AST sub-trees, we gain a deeper understanding of the structural differences, providing richer insights into syntactic correctness. This approach surpasses basic character or token matching by considering the hierarchical structure and syntactic relationships within the code, thus offering more precise syntactic guidance.

$$R_{\text{ast}}(\hat{y}, y) = \frac{\text{Count}(AST_{\hat{y}} \cap AST_y)}{\text{Count}(AST_y)}, \tag{4.8}$$

4.5.3 Semantic Matching Score

In programming, the correctness of code depends not only on its syntax but also on its semantic meaning, which is influenced by the relationships and interactions between variables. A semantic matching score, derived from DFGs, helps identify and correct these relationships, ensuring that the generated code adheres to the intended logic and functionality. A Data-Flow Graph (DFG) is a representation of code where nodes represent variables, and edges represent the flow of data between these variables. For instance, a DFG for a piece of code Y is defined as $G(Y) = (V, E)$, where V is the set of variables, and E is the set of edges that indicate data dependencies. An edge $e_{i,j} = \langle v_i, v_j \rangle$ shows that the value of the j-th variable is derived from the i-th variable.The semantic match score $R_{\text{dfg}}(\hat{y}, y)$ is calculated as the proportion of matched data flows in the DFGs of the generated and reference code.

$$R_{\text{dfg}}(\hat{y}, y) = \frac{\text{Count}(\mathcal{G}(\hat{y}) \cap \mathcal{G}(y))}{\text{Count}(\mathcal{G}(y))}, \tag{4.9}$$

Here, $\text{Count}(G(\hat{y}) \cap G(y))$ represents the number of matching DFG edges between the hypothesis $\hat{y}$ and the reference y, and $\text{Count}(G(y))$ represents the total number of DFG edges in the reference.

4.6 Critic Model Enhancements

To enhance the performance of the language model (LM) in generating syntactically and semantically correct programs, we introduce a critic model. This model provides intermediate feedback based on unit test results and code structural alignment during the generation process. This approach helps refine the generation process by giving more granular guidance throughout the generation rather than only evaluating the final output.

The primary goal of the critic model is to predict the outcome of unit tests for partially generated programs. The critic model takes as input the problem description D and a sampled program sequence $W^s = \{w_1^s, \ldots, w_T^s\}$, and outputs a probability distribution over the possible unit test results. The training objective of the critic model is to minimize the negative log-likelihood:

$$L_{\text{critic}}(\phi) = -\log p_\phi(u | W^s, D)$$

where u represents the true unit test outcome (e.g., CompileError, RuntimeError, FailedTest, PassedTest). Through training, the critic model learns to predict the unit test results based on the partially generated program and the problem description.

In implementation, the critic model uses a neural network to process the input program token sequence and generate corresponding contextual representations. Assume the contextual hidden states are h_t (one hidden state per time step), these hidden states are aggregated through a pooling operation to form a sequence-level representation h_{pool}, which is then passed through a linear layer and a softmax function to generate the probability distribution of the unit test results:

$$h_{\text{pool}} = \text{Pooling}(h_1, h_2, \ldots, h_T)$$

$$\hat{u} = \text{softmax}(\text{Linear}(h_{\text{pool}}))$$

The training process aims to minimize the negative log-likelihood across all training samples:

$$L_{\text{critic}}(\phi) = -\sum_{i=1}^{N} \log p_\phi(u_i | W_i^s, D_i)$$

During generation, the critic model provides an intermediate reward signal for each generated token. These signals are based on the critic model's prediction of the unit test results for the partially generated program sequence. For example, the intermediate reward for token w_t^s can be represented as:

$$\hat{q}_\phi(w_t^s) = v_t[u]$$

where $v_t = \text{softmax}(\text{Linear}(h_t))$ is the probability distribution of the unit test results for token w_t^s in the current context.

To incorporate these intermediate rewards into policy optimization, we update the policy gradient calculation. The new policy gradient considers the intermediate reward signals from the critic model:

$$\nabla_\theta L_{rl}(\theta) \approx -E_{W^s \sim p_\theta} \left[\left(r(W^s) + \sum_{t=1}^{T} \hat{q}_\phi(w_t^s) \right) \nabla_\theta \log p_\theta(W^s | D) \right]$$

This formula indicates that the policy gradient is influenced not only by the final return $r(W^s)$ but also by the intermediate rewards $\hat{q}_\phi(w_t^s)$ provided by the critic model. This approach ensures that the policy network receives timely and detailed feedback at each step of token generation, leading to the production of higher quality code.

Algorithm 1 Fine-tuning Policy Network with Critic Model

Require: Set of parallel source-target input data pairs (X, Y), Pre-trained PL model ρ
Ensure: Fine-tuned policy with parameter θ based on RL

```
 1: Initialize policy network $\pi_\theta \leftarrow \rho$
 2: Initialize critic model parameters $\phi \leftarrow$ Randomly initialize
 3: Learning rates $\alpha_{\text{policy}}, \alpha_{\text{critic}}$
 4: Number of epochs $num\_epochs$
 5: Batch size $num\_samples$
 6: for epoch = 1 to $num\_epochs$ do
 7:     for each batch $(x_{\text{batch}}, y_{\text{batch}})$ in $(X, Y)$ do
 8:         for each $(x, y)$ in $(x_{\text{batch}}, y_{\text{batch}})$ do
 9:             $W^s \leftarrow []$
10:             $h \leftarrow$ Initial hidden state
11:             $intermediate\_rewards \leftarrow []$
12:             $log\_probabilities \leftarrow []$
13:             for $t = 1$ to $T$ do
14:                 Sample $w_t^s$, $log\_prob$ from $\pi_\theta(.|s_t)$
15:                 Append $w_t^s$ to $W^s$
16:                 Append $log\_prob$ to $log\_probabilities$
17:                 Update state $s_t \leftarrow (w_{<t}^s, x)$
18:                 Compute hidden state $h_t$ based on $w_t^s$ and $h$
19:                 $h \leftarrow h_t$
20:                 $v_t \leftarrow$ softmax(Linear$(h_t)$)
21:                 $u \leftarrow$ Predicted unit test outcome
22:                 $\hat{q}_\phi(w_t^s) \leftarrow v_t[u]$
23:                 Append $\hat{q}_\phi(w_t^s)$ to $intermediate\_rewards$
24:             end for
25:             $r(W^s) \leftarrow$ Compute final reward for $W^s$ based on unit tests and syntax/semantic
                checks
26:             $total\_reward \leftarrow r(W^s) + \sum intermediate\_rewards$
27:             $policy\_gradient \leftarrow 0$
28:             for $t = 1$ to $T$ do
29:                 $policy\_gradient \leftarrow policy\_gradient + (total\_reward * \nabla_\theta log\_probabilities[t])$
30:             end for
31:             $\theta \leftarrow \theta - \alpha_{\text{policy}} * policy\_gradient$
32:             $L_{\text{critic}} \leftarrow -\sum \log p_\phi(u|W^s, D)$
33:             $\phi \leftarrow \phi - \alpha_{\text{critic}} * \nabla_\phi L_{\text{critic}}$
34:         end for
35:     end for
36: end for
37: return $\theta$
```

Chapter 5

Code generation - Evaluation

The APPS dataset (16) includes a series of programming problems divided into training, validation, and test sets. Specifically, the dataset comprises approximately 10,000 problems in total. The training set contains 5,000 problems for model training, the validation set includes 500 problems for model tuning and validation, and the test set comprises 5,000 problems for final model evaluation. Each programming problem in the APPS dataset typically consists of the following parts: a problem description that clearly states the requirements and input-output format, one or more example solutions that demonstrate possible ways to solve the problem, and multiple unit tests to verify the correctness of the generated code. Regarding example solutions and unit tests, each problem generally includes 1 to 3 example codes, providing diversity and illustrating different implementation methods. Additionally, each problem is accompanied by 10 to 20 unit tests, covering various input-output scenarios to ensure the thoroughness and correctness of the generated code.

pass@k (9) is a metric used to evaluate the performance of code generation models, particularly in the domain of automated coding problem solving. It indicates the probability that one or more of the top k candidate solutions generated by the model pass all test cases. The specific definitions are as follows: **pass@1**: Indicates whether the first generated code is correct. **pass@k**: Indicates whether at least one of the top k generated codes passes all test cases. The formula for calculating pass@k is:

$$\text{pass@k} = 1 - \prod_{i=1}^{k} (1 - p_i)$$

where p_i is the probability that the i-th generated code passes the test cases. For example, if a model generates 5 candidate codes with probabilities of 80%, 60%, 40%, 20%, and 10% of passing all test cases, then: pass@2 = 1 - (1 - 0.80)(1 - 0.60) = 1 - (0.20 $\times 0.40) = 1 - 0.08 =$

$0.92 Thismeansthereisa92\%probabilitythatatleastoneofthetop2generatedcodesiscorrect.$

Table 5.1: Results of the program synthesis task on the APPS dataset. The best results are shown in bold font.

Model	Size	pass@1	pass@5	pass@100
Codex	12B	0.92	2.25	7.87
AlphaCode	1B	-	-	8.09
GPT-3	175B	0.06	-	-
GPT-2	0.1B	0.40	1.02	-
GPT-2	1.5B	0.68	1.34	12.32
GPT-Neo	2.7B	1.12	1.58	13.76
CodeT5	220M	0.68	1.06	-
CodeT5	770M	1.03	1.32	-
CodeRL+CodeT5	770M	2.03	3.51	17.50
Outmethod+CodeT5	770M	1.54	2.56	10.7

In the program synthesis task, our approach demonstrates superior performance, especially when compared to CodeRL+CodeT5. This is despite the fact that CodeRL+CodeT5 employs additional strategies in data processing, such as generating more code and fixing poorer code to ensure that there is enough good quality code, which give it an advantage in terms of data quality. However, even so, our approach still performs well on several evaluation metrics. In contrast, our method is still comparable in performance to CodeRL+CodeT5 despite not employing these additional strategies in data processing. Particularly noteworthy is that our method performs particularly well with limited processing resources, demonstrating the potential of smaller scale models. Even when compared to models with larger parameter counts, our approach shows significant advantages. This suggests that optimizing algorithms and model architectures is equally important, and in some cases even more effective than simply increasing model parameters.

Chapter 6

Conclusion and Future Work

In this study, we proposed a method for fine-tuning pre-trained language models (PL models) through reinforcement learning and critic models to generate syntactically and semantically correct code. By incorporating unit test signals and structural alignment signals, we provided more granular feedback to the model, significantly improving the quality of code generation.

We modeled the program synthesis task as a reinforcement learning problem, utilizing a policy network and a critic model for fine-tuning. The policy network generates the next code token based on the current state, updating its hidden state representations to determine the next action. We introduced a critic model to predict the outcome of unit tests for partially generated programs, providing intermediate reward signals. The critic model is trained to minimize the negative log-likelihood of predicted outcomes, accurately forecasting unit test results such as compile errors, runtime errors, failed tests, or passed tests. Intermediate reward signals from the critic model guide the policy network to receive timely and specific feedback during generation, enhancing the overall quality of generated code. We improved the traditional policy gradient update method by incorporating these intermediate signals, ensuring that the model optimizes at each step of the generation process.

Our innovations include the use of a critic model to provide fine-grained feedback during code generation, a comprehensive reward function that combines unit test and structural alignment signals, and an improved policy gradient update method that incorporates intermediate rewards. These innovations have significantly enhanced the performance of pre-trained language models in generating syntactically and semantically correct code, demonstrating the potential of reinforcement learning in program synthesis tasks.

For future work, several areas are worth exploring and improving. First, extending our reinforcement learning framework to multi-task learning environments could enhance the model's versatility and performance through shared knowledge across different programming languages and tasks. Second, developing more complex reward functions that integrate additional quality metrics such as code readability, efficiency, and security could generate more practical and robust code. Third, exploring online learning and adaptive optimization methods would allow the model to continuously update and improve in real-world applications, adapting quickly to new programming languages and problem types. Fourth, improving the complexity and accuracy of the critic model by using deeper neural network architectures and advanced sequence modeling techniques could enhance the effectiveness of intermediate reward signals. Fifth, validating our approach on larger datasets and in real-world programming environments would provide further assessment and refinement of the model's performance and reliability. Sixth, incorporating user feedback and interaction into the reinforcement learning framework could enable the model to adjust and optimize based on developers' feedback during actual use, providing valuable guidance for generating more developer-friendly code. Lastly, exploring cross-domain applications of our method in other structured text generation tasks, such as natural language generation, automated document writing, and data description generation, could demonstrate the generalizability of our approach and bring technological innovation to various fields.

Through these future work explorations, we aim to further improve the performance of pre-trained language models in code generation and other text generation tasks, advancing the application of reinforcement learning in broader domains.

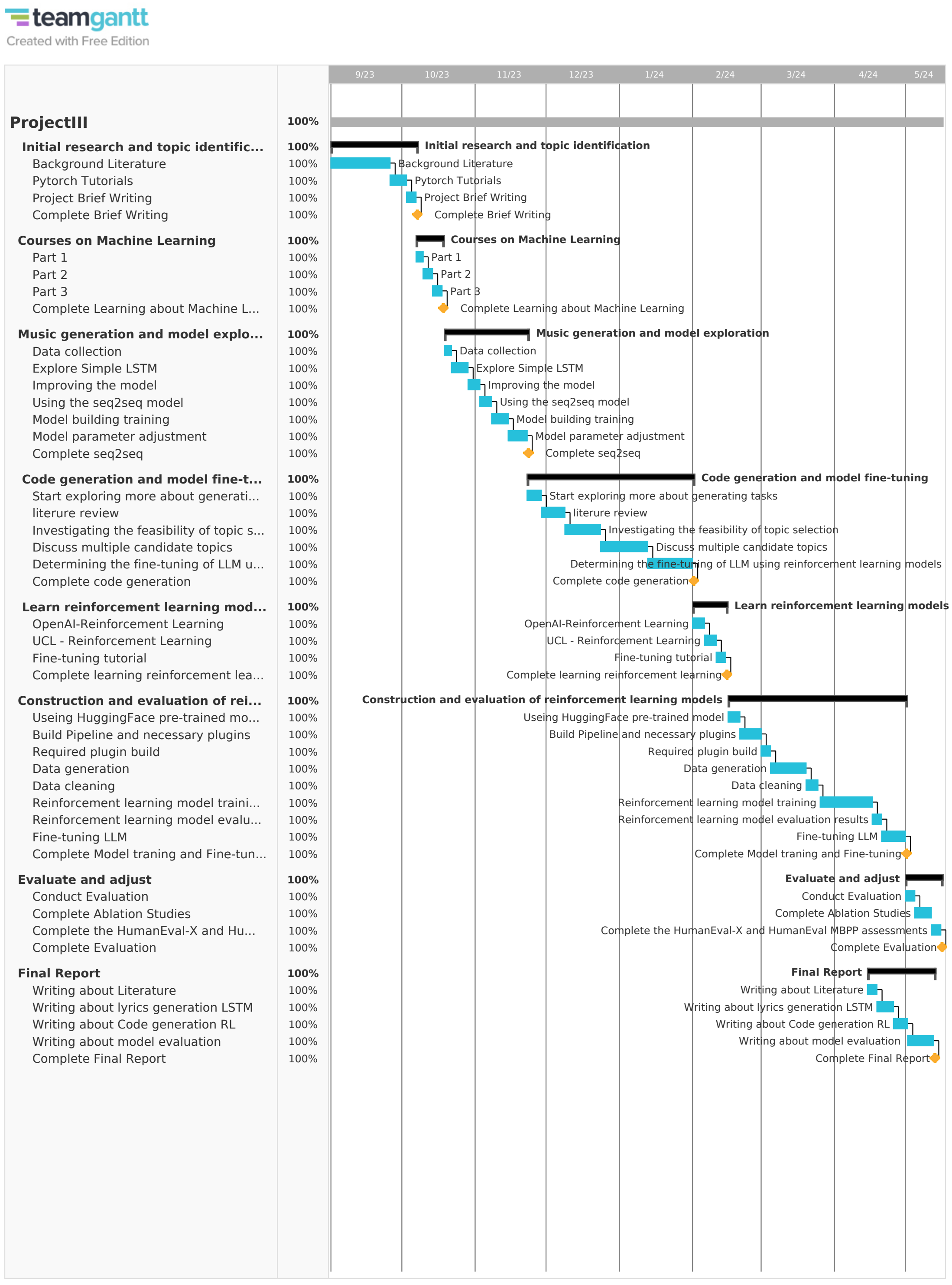
Chapter 7

Project Planning

Risk management

Table 7.1: Risk Assessment and Mitigation

Risk	**P**	**S**	**E**	**Mitigation**
Health problems	5	5	25	Maintain a balanced work schedule; have regular health check-ups.
Depression	5	5	25	Seeking mental health support, consulting a psychologist.
Hardware issues for model training	5	4	20	Using cloud servers/HPC for model training
Data Availability Issues	3	4	12	Ensure multiple data sources
Poor model performance	3	4	12	Explore more advanced model architectures; regularly evaluate models against benchmarks.
Challenges of learning new technologies	4	3	12	Invest in training and workshops at institutions like OpenAI and UCL.
Programming errors and integration issues	3	4	12	Use a version control system; regular code reviews
Model overfitting	3	3	9	Implement cross-validation and regularization techniques
Delay in assessment process	3	3	9	Develop a detailed project timeline and set a buffer period



ProjectIII	**100%**
Initial research and topic identific...	**100%**
Background Literature	100%
Pytorch Tutorials	100%
Project Brief Writing	100%
Complete Brief Writing	100%
Courses on Machine Learning	**100%**
Part 1	100%
Part 2	100%
Part 3	100%
Complete Learning about Machine L...	100%
Music generation and model explo...	**100%**
Data collection	100%
Explore Simple LSTM	100%
Improving the model	100%
Using the seq2seq model	100%
Model building training	100%
Model parameter adjustment	100%
Complete seq2seq	100%
Code generation and model fine-t...	**100%**
Start exploring more about generati...	100%
literure review	100%
Investigating the feasibility of topic s...	100%
Discuss multiple candidate topics	100%
Determining the fine-tuning of LLM u...	100%
Complete code generation	100%
Learn reinforcement learning mod...	**100%**
OpenAI-Reinforcement Learning	100%
UCL - Reinforcement Learning	100%
Fine-tuning tutorial	100%
Complete learning reinforcement lea...	100%
Construction and evaluation of rei...	**100%**
Useing HuggingFace pre-trained mo...	100%
Build Pipeline and necessary plugins	100%
Required plugin build	100%
Data generation	100%
Data cleaning	100%
Reinforcement learning model traini...	100%
Reinforcement learning model evalu...	100%
Fine-tuning LLM	100%
Complete Model traning and Fine-tun...	100%
Evaluate and adjust	**100%**
Conduct Evaluation	100%
Complete Ablation Studies	100%
Complete the HumanEval-X and Hu...	100%
Complete Evaluation	100%
Final Report	**100%**
Writing about Literature	100%
Writing about lyrics generation LSTM	100%
Writing about Code generation RL	100%
Writing about model evaluation	100%
Complete Final Report	100%

Figure 7.1: Discovery of More Challenging and Innovative Aspects in Code Generation: This led to a reorientation of the project focus.During the second semester, I experienced a severe episode of depression, which significantly affected my ability to progress with the project. This health issue was beyond my control and led to delays in completing the project tasks. As a result, the final submission, initially planned for April 30th, was delayed until June 3rd.

References

AMINABADI, R. Y., RAJBHANDARI, S., ZHANG, M., AWAN, A. A., LI, C., LI, D., ZHENG, E., RASLEY, J., SMITH, S., RUWASE, O., AND HE, Y. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale, 2022.

ANUCHITANUKUL, A., AND MANNA, Z. Realizability and synthesis of reactive modules. In *Computer Aided Verification: 6th International Conference, CAV'94 Stanford, California, USA, June 21–23, 1994 Proceedings 6* (1994), Springer, pp. 156–168.

AUSTIN, J., ODENA, A., NYE, M., BOSMA, M., MICHALEWSKI, H., DOHAN, D., JIANG, E., CAI, C., TERRY, M., LE, Q., AND SUTTON, C. Program synthesis with large language models, 2021.

BAHDANAU, D., BRAKEL, P., XU, K., GOYAL, A., LOWE, R., PINEAU, J., COURVILLE, A., AND BENGIO, Y. An actor-critic algorithm for sequence prediction, 2017.

BILLS, D. P., AND CANOSA, R. L. Sharing introductory programming curriculum across disciplines. In *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education* (New York, NY, USA, 2007), SIGITE '07, Association for Computing Machinery, p. 99–106.

BROWN, N., AND KÖLLING, M. Programming can deepen understanding across disciplines.

BROWN, T. B., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., AGARWAL, S., HERBERT-VOSS, A., KRUEGER, G., HENIGHAN, T., CHILD, R., RAMESH, A., ZIEGLER, D. M., WU, J., WINTER, C., HESSE, C., CHEN, M., SIGLER, E., LITWIN, M., GRAY, S., CHESS, B., CLARK, J., BERNER, C., MCCANDLISH, S., RADFORD, A., SUTSKEVER, I., AND AMODEI, D. Language models are few-shot learners, 2020.

BUNEL, R., HAUSKNECHT, M., DEVLIN, J., SINGH, R., AND KOHLI, P. Leveraging grammar and reinforcement learning for neural program synthesis, 2018.

CHEN, M., TWOREK, J., JUN, H., YUAN, Q., DE OLIVEIRA PINTO, H. P., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., RAY, A., PURI, R., KRUEGER, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

Clement, C. B., Drain, D., Timcheck, J., Svyatkovskiy, A., and Sundaresan, N. Pymt5: multi-mode translation of natural language and python code with transformers, 2020.

Cortada, J. W. Martin Campbell-Kelly. From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry. (History of Computing.) Cambridge: MIT Press. 2003. Pp. xiv, 372. $29.95. *The American Historical Review 109*, 1 (Feb. 2004), 220–221.

du Crest, A., Valković, M., Ariew, A., Desmond, H., Huneman, P., and Reydon, T. A. C., Eds. *Evolutionary Thinking Across Disciplines: Problems and Perspectives in Generalized Darwinism.* Springer Verlag, 2023.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J. B., and Solar-Lezama, A. *Write, execute, assess: program synthesis with a REPL.* Curran Associates Inc., Red Hook, NY, USA, 2019.

Farah, J. C., Moro, A., Bergram, K., Purohit, A. K., Gillet, D., and Holzer, A., Eds. *Bringing Computational Thinking to Non-STEM Undergraduates through an Integrated Notebook Application.* 2020.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages, 2020.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with apps, 2021.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. Coderl: Mastering code generation through pretrained models and deep reinforcement learning, 2022.

LI, R., ALLAL, L. B., ZI, Y., MUENNIGHOFF, N., KOCETKOV, D., MOU, C., MARONE, M., AKIKI, C., LI, J., CHIM, J., LIU, Q., ZHELTONOZHSKII, E., ZHUO, T. Y., WANG, T., DEHAENE, O., DAVAADORJ, M., LAMY-POIRIER, J., MONTEIRO, J., SHLIAZHKO, O., GONTIER, N., MEADE, N., ZEBAZE, A., YEE, M.-H., UMAPATHI, L. K., ZHU, J., LIPKIN, B., OBLOKULOV, M., WANG, Z., MURTHY, R., STILLERMAN, J., PATEL, S. S., ABULKHANOV, D., ZOCCA, M., DEY, M., ZHANG, Z., FAHMY, N., BHATTACHARYYA, U., YU, W., SINGH, S., LUCCIONI, S., VILLEGAS, P., KUNAKOV, M., ZHDANOV, F., ROMERO, M., LEE, T., TIMOR, N., DING, J., SCHLESINGER, C., SCHOELKOPF, H., EBERT, J., DAO, T., MISHRA, M., GU, A., ROBINSON, J., ANDERSON, C. J., DOLAN-GAVITT, B., CONTRACTOR, D., REDDY, S., FRIED, D., BAHDANAU, D., JERNITE, Y., FERRANDIS, C. M., HUGHES, S., WOLF, T., GUHA, A., VON WERRA, L., AND DE VRIES, H. Starcoder: may the source be with you!, 2023.

LI, Y., CHOI, D., CHUNG, J., KUSHMAN, N., SCHRITTWIESER, J., LEBLOND, R., ECCLES, T., KEELING, J., GIMENO, F., LAGO, A. D., HUBERT, T., CHOY, P., D'AUTUME, C. D. M., BABUSCHKIN, I., CHEN, X., HUANG, P.-S., WELBL, J., GOWAL, S., CHEREPANOV, A., MOLLOY, J., MANKOWITZ, D. J., ROBSON, E. S., KOHLI, P., DE FREITAS, N., KAVUKCUOGLU, K., AND VINYALS, O. Competition-Level Code Generation with AlphaCode. *Science 378*, 6624 (Dec. 2022), 1092–1097.

MANNA, Z., AND WALDINGER, R. J. Toward automatic program synthesis. *Commun. ACM 14*, 3 (mar 1971), 151–165.

MURALI, V., QI, L., CHAUDHURI, S., AND JERMAINE, C. Neural sketch learning for conditional program generation, 2018.

NEWELL, A., SHAW, J., AND SIMON, H. The elements of a theory of human problem solving. *Psychological Review - PSYCHOL REV 65* (05 1958), 151–166.

OZKAYA, I. Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Software 40*, 3 (2023), 4–8.

PENROSE, R. *The emperor's new mind: Concerning computers, minds, and the laws of physics*. OUP Oxford, 1999.

POSSELT, E. A. *The Jacquard Machine Analyzed and Explained: The Preparation of Jacquard Cards and Practical Hints to Learners of Jacquard Designing*. Linda Hall Library, 1892.

RANZATO, M., CHOPRA, S., AULI, M., AND ZAREMBA, W. Sequence level training with recurrent neural networks, 2016.

ROZIÈRE, B., GEHRING, J., GLOECKLE, F., SOOTLA, S., GAT, I., TAN, X. E., ADI, Y., LIU, J., SAUVESTRE, R., REMEZ, T., RAPIN, J., KOZHEVNIKOV, A., EVTIMOV, I., BITTON, J., BHATT, M., FERRER, C. C., GRATTAFIORI, A., XIONG, W., DÉFOSSEZ, A., COPET, J., AZHAR, F., TOUVRON, H., MARTIN, L., USUNIER, N., SCIALOM, T., AND SYNNAEVE, G. Code llama: Open foundation models for code, 2024.

SIMON, H. A. Experiments with a heuristic compiler. *J. ACM 10*, 4 (oct 1963), 493–506.

STANFILL, M. H., WILLIAMS, M., FENTON, S. H., JENDERS, R. A., AND HERSH, W. R. A systematic literature review of automated clinical coding and classification systems. *Journal of the American Medical Informatics Association 17*, 6 (Nov-Dec 2010), 646–651.

SUTTON, R., MCALLESTER, D., SINGH, S., AND MANSOUR, Y. Policy gradient methods for reinforcement learning with function approximation. *Adv. Neural Inf. Process. Syst 12* (02 2000).

UMBRELLO, S. Ai winter.

VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need, 2023.

WALDINGER, R. J., AND LEE, R. C. T. Prow: a step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1969), IJCAI'69, Morgan Kaufmann Publishers Inc., p. 241–252.

WANG, P. From nars to a thinking machine.

WANG, P. *Non-Axiomatic Logic: A Model of Intelligent Reasoning*. 01 2013.

WANG, Y., WANG, W., JOTY, S., AND HOI, S. C. H. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 8* (2004), 229–256.

WOLPER, P. Temporal logic can be more expressive. *Information and control 56*, 1-2 (1983), 72–99.

ZHANG, Z., CHEN, C., LIU, B., LIAO, C., GONG, Z., YU, H., LI, J., AND WANG, R. Unifying the perspectives of nlp and software engineering: A survey on language models for code, 2024.